(4)

Multiprogramming and the Performance of Parallel Programs

by

Muhammad S. Benten and Harry F. Jordan

Computer Systems Design Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309-0425

CSDG 88-2
January 1988

BIBLIOGRAPHIC DATA SHEET	1. Report No. ECE Technical Report 88-1-2	2.	3. Recipient's Accession No.
4. Title and Subtitle Multiprogramming and the Performance of Parallel Programs			5. Report Date January 1988
			6.
7. Author(s) Muhammad S. Benten and Harry F. Jordan			8. Performing Organization Rept. No. CSDG 88-2
9. Performing Organization Name and Address Computer Systems Design Group, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309-0425			10. Project/Task/Work Unit No.
			11. Contract/Grant No. N00014-86-k-0204
12. Sponsoring Organization Name and Address Office of Naval Research, 800 N. Quincy Street, Arlington, VA 22217-5000			13. Type of Report & Period Covered Interim
			14.

15. Supplementary Notes

Also supported in part by NASA Langley Research Center under grant NAG-1-640.

16. Abstracts

Tight synchronization in parallel programs executed on multiprogrammed multiprocessors may result in catastrophic performance losses as a result of the absence of swapped out processes. Our work introduces a programming methodology that utilizes computational synchronization and avoids tight control flow synchronization in parallel programs. In this methodology, each phase of the computation is assigned a status that can be ready, blocked or completed, and tasks in each computational phase are selfscheduled to ensure computational progress by the available executing processes. Results obtained indicate that this methodology avoids the catastrophic performance losses resulting from the swapping of processes in multiprogrammed multiprocessors.

17. Key Words and Document Analysis. 17a. Descriptors

multiprocessor
multiprogramming
shared memory
performance
synchronization

17b. Identifiers/Open-Ended Terms

the Force

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 11
	20. Security Class (This Page) UNCLASSIFIED	22. Price

Abstract Tight synchronization in parallel programs executed on multiprogrammed multiprocessors may result in catastrophic performance losses as a result of the absence of swapped out processes. Our work introduces a programming methodology that utilizes computational synchronization and avoids tight control flow synchronization in parallel programs. In this methodology, each phase of the computation is assigned a status that can be ready, blocked or completed, and tasks in each computational phase are selfscheduled to ensure computational progress by the available executing processes. Results obtained indicate that this methodology avoids the catastrophic performance losses resulting from the swapping of processes in multiprogrammed multiprocessors.

1. Introduction Recent advances in hardware technologies and the advances in parallel languages and algorithms are promising to make shared memory multiprocessors the future computation machines. Currently, there are many shared memory multiprocessors available, ranging from supercomputers to the minisuper and superminicomputers, and all have proven to be able to deliver very high throughputs that satisfy time sharing and batch programs needs. It has also been seen that these machines are capable of supplying speedup of single parallel programs when their execution is implemented by a set of cooperating processes. However, most of these machines run multiprogrammed operating systems and thus there is no certainty that processes of a parallel program will be executed concurrently on more than one hardware processor. In many instances, multiprogramming, which is a property of the operating system aimed towards the improvement of the overall system efficiency and throughput, interferes with parallel programs, and slowdowns in the execution time of these programs occur.

In this paper we will be discussing the effects of multiprogramming on the performance of parallel programs, where the performance of a parallel program is determined in terms of the wall clock time speed up of executing the parallel program over the sequential version. We will show this effect by presenting the results of executing an example parallel program on three multiprocessors; an Alliant FX/8, a 20 processor Encore Multimax and an 8 processor Sequent Balance 21000. These results will be given for situations where the parallel program processes were executed concurrently on separate processors and for situations where they were multiprogrammed on a smaller set of CPU's. In a later section, we will also present a new programming

This work was supported in part by NASA Langley Research Center under NAG-1-640, and by the Office of Naval Research under N00014-86-K-0204.

methodology that will improve the performance of parallel programs and is suitable for multiprogrammed environments.

Throughout this paper we will assume the use of a parallel language, where parallelism is at the top of the program hierarchy, in particular we will be using the Force[1]. The Force is a parallel language that is currently being developed at the University of Colorado, Boulder, and is currently implemented on several machines. Among these are Flex's, Cray's, Alliant's, Sequent's and Encore's machines. In the Force and similar parallel programming paradigms, such as IBM EPEX Fortran[2], and Butterfly Uniform System library[3], the user writes a single program that is to be executed by an arbitrary number of processes, where the number is not specified until run time and stays fixed throughout the execution period. These processes are the parallel instruction streams that will participate in the execution of the parallel program. In the parallel programmer's view, these streams are supposed to be executed concurrently on separate CPU's during the execution of his parallel program. However, as a result of multiprogramming, coscheduling of these processes may not be possible and the performance of parallel programs may be severely affected.

2. Example Parallel Program Consider the LU factorization of a matrix with partial pivoting as implemented in the Linpack SGEFA subroutine[4]. The algorithm is composed of three phases that are repeated for all rows of the matrix to be factored. These phases must be completed in order for each row, before they can start for the next one. The search for the pivot row is done in the first phase and the swapping of the pivot row and the current row is done in the second phase. In the third phase, the actual Gaussian elimination computations are performed. Although, there are other papers that consider the possibility of overlapping computational phases[5], our aim in this paper is to show the effects of multiprogramming on tightly synchronized and strictly ordered phases.

Using the Force, this algorithm was written by Jordan[6], as a Force subroutine. This Force subroutine starts with a barrier[8], which is a Force construct that is used to synchronize the instruction streams. The barrier will let the force of processes executing a Force program wait until they all execute the barrier statement. One process of the Force can then execute a sequential code section, if any, and the Force is released again. The barrier provides a very clear and simple way to mark the completion of a computational phase and the availability of all the processes to start the next phase of the computation. The barrier in this algorithm initializes shared variables before the factorization process starts. The barrier is followed by the main loop over the rows of the matrix. In this loop the first phase is done using a prescheduled Do loop[9], followed by a critical section. The second phase consists of swapping the pivot row and the current row if swapping is necessary. Obviously, this can not be started before the pivot row is located, hence, a Force barrier is used to detect the completion of the first phase by detecting the arrival of all the processes that participated in finding the pivot. The third phase is that of performing the actual row reductions. Again the third phase is separated from the second phase by a barrier which will also calculate the pivot element required for the elimination. The Force subroutine which implements the SGEFA algorithm is shown in Table-I.

2.1 Performance of the LU example The results of executing the LU factorization algorithm on an Encore's Multimax, Alliant's FX/8 and Sequent's Balance 21000 is shown in Figures (1,2,3), respectively. In these figures, the light bars show the results for the case where coscheduling of processes on separate CPU's is guaranteed and multiprogramming is avoided by making sure that no other jobs are running on the machine when this program is run. Clearly, these results are excellent, and almost linear speedups are obtained, up to the number of processors in these machines. This algorithm, which performs and speeds up very well, is executed by all the processes in a lock step fashion, and the coscheduling of processes involved in the computation on separate processors is required to obtain this performance. When there are fewer processors available than processes, these processes will be multiprogrammed on the available processors. The performance degradation due to multiprogramming can be seen from the previous figures when the number of processes exceeded the number of CPU's. Although, the user usually restricts the number of processes he uses to be less or equal to the number of hardware processors available, in an environment like Unix, he can't prevent other users or system processes from running on the machine, and thus, these processes will be multiprogrammed with the processes of his parallel program.

Due to multiprogramming, which is the scheduling of a set of processes on a smaller set of processors, some of the processes will be suspended in the middle of their execution. In this environment, the performance of the LU algorithm is shown by the dark bars in Figures(1,2,3). In these figures, multiprogramming is manifested by the deterioration in the performance of the parallel algorithm and the considerable slow down that has been incurred. Although this effect varies for the machines that have been used, these variations are related to memory and cacheing strategies as well as to operating system's tuning parameters such as the length of the time quantum and context switching overhead on each machine. The locking mechanism also plays a very important role and since it is based on spin locks on these machines, processes waiting for locks will be consuming their time slices which results in a lot of wasted CPU cycles. Thus, in tightly synchronized parallel programs, suspended process may preclude the advancement of other executing processes and limit progress in the overall program.

The light bars in figures (1,2,3), represent the execution time when the machines were not loaded, and improvement in the performance is obtained up to the number of physical processors on these machines. When running with more than 8 processes on Alliant's and Sequent's machines, the performance started to deteriorate and by 9 processes the performance is slower than running with one process. On the Encore, running with 24 processes is as slow as running with one process. The dark bars, show the performance of Jordan's LU Factorization algorithm on a loaded machine. In this case, the algorithm shows improvements up to 5 processes on the Multimax and up to 3 processes on the Sequent and and the Alliant, above these numbers the performance is deteriorating. Evidently, a main cause of this degradation is synchronization. Critical sections are one of the causes since a suspended process that has a lock will block others waiting for that lock. However, barriers are the major cause as the barrier would force the executing processes to wait for each other. In fact, the execution time of a parallel code section that is enclosed between two barriers or equivalent stream synchronizing constructs in a parallel program, is limited by the time required for the slowest process to enter the first barrier, execute through the code section and exit the second barrier. The barrier construct is a source of degradation not only due to the time spent waiting for suspended processes, but also due to the critical sections that are used in its implementation and involve all processes[8].

3. A New Parallel Programming Methodology In this section, we are going to present a new programming methodology that can be used to design and implement parallel algorithms on shared memory multiprocessors. Algorithms which are to be implemented using this methodology are assumed to be decomposable into phases that have to be completed in order. This methodology will make no assumption about the number or the speed of processes that will participate in the completion of a computation. The completion of computations and progress made in them will be decided primarily by the status of the computation and will be less dependent on the number of executing processes. The key issue in this methodology is the use of work barriers to detect the completion of previous phases rather than the use of the traditional stream barriers. The avoidance of stream barriers will also limit synchronization to the access of shared data, and will let the available processes proceed as long as they are not blocked by locks acquired by suspended processes.

Given a computation that consists of a number of subcomputations that have to be completed in order, each subcomputation will referred to as a computational phase. In general, a computational phase represents a computation and a synchronization structure that surrounds the computation to insure its scheduling and completion regardless of the number of processes in a parallel program that will execute or pass over it. Currently, computations which can be handled by this construct are assumed to consist of a group of parallel tasks which are represented by indices that cover a range of integers, similar to those problems that can be implemented using DOALL loops[9]. An optional critical code section can also be placed following the computation and will be executed by every process that entered the computational phase and executed the DOALL loop. The last section of a computational phase is a strictly sequential section of code that is referred to as the completion section of the computational phase. It is executed by one process after the computation has been completed.

A computational phase has a status that can be blocked, ready, or completed. When the status of a computational phase is found to be blocked (not ready), processes that try to execute this computational phase will wait until the status is set to unblocked (ready) by some other process. When it is unblocked (ready), the construct will insure that no process can proceed with the parallel program section following the computational phase before all the tasks in this subcomputation have been scheduled. Processes can enter a computational phase if it is unblocked, and will execute in it if its computation and its completion section have not been finished, otherwise, the computational phase will automatically become passive as its status will be marked as completed. Processes that encounter a computational phase that has been completed, will skip it and continue with the statement or construct that follows the computational phase. The completion section of the current phase and the status of the following phase together comprise a work barrier if the completion section of the current phase contains a statement that would mark the next blocked phase as ready. This will insure the integrity of the computation without the need for waiting for all processes at the end of each phase.

3.1 LU factorization and the New Methodology Utilizing the concept of a computational phase, the LU factorization algorithm was rewritten as a subroutine in an extension version of the Force. The algorithm starts by initializing shared memory using a structure referred to as the *Init* construct. This construct is characterized by blocking processes until the initialization body has been executed by a single process which succeeded in obtaining a shared lock before any other process could. Processes arriving after the initialization code has been executed will skip the body of the *Init* construct and proceed with the code following the *Init*. The structure of this construct is as follows:

```
Each process will:
    Atomically do:
        - check if init done,
          if done then skip the next two steps.
        - do initialization.
        - mark init done.
```

The use of this construct in initialization will allow the section of code to be executed before any process can proceed and will not wait for all the processes to arrive. This construct will be reinitialized in an *Init* construct at the end of this subroutine so that other calls to this subroutine will execute correctly. The *Init* construct we are using for Gaussian elimination will do the following initializations:

(1) Initialize the shared row marker (pivot row) to 1, the first row.

(2) Initialize the status of the pivot search computational phase to *ready*.

(3) Reinitialize the *Init* construct at the end of this Force subroutine so that the first process to exit this subroutine will execute it.

(4) Finally, the process executing this construct will wait for the matrix to be factored to become ready.

The *Init* construct, in this algorithm, is followed by an iterative loop which is to be executed in parallel (n-1) times, where n is the number of rows in the matrix to be factored.

Processes arriving at the beginning of the main algorithm loop, after passing by the *init* construct, can start the LU factorization process because the matrix has been set up, the working row has been initialized, and no process will get into the previous *Init* construct to change the initialized shared information. The main body loop consists of three computational phases, the pivot search phase, row swapping phase and the row reduction phase. This algorithm .. written such that one phase will be ready at a time while the other phases are blocked. Initially all phases are blocked except for the pivot search phase. A block of strictly sequential code terminating each phase will block the current phase and mark the following one as ready. At first, processes can only execute the pivot search section which is described below.

Processes that succeed in getting into the pivot search computational phase will be self scheduled to search for the pivot element, each process will search for the row with the maximum pivot element among the rows it is assigned. Processes will then execute the critical section that follows the DOALL loop, so that the global maximum pivot row is identified, and will proceed to the completion section of the phase. The body of this section is to be executed by the first process that reaches it after the computation has been completed. Early as well as late arriving processes will skip this section and proceed to the next phase. The body of this section consists of blocking this phase (pivot search), setting up the next phase (row swapping), doing any required initializations and marking it as ready. When this is done, processes waiting for the row swapping computational phase will be let go, and every other phase would have been blocked.

The row swapping phase is organized in a way similar to the pivot search phase. Its computational section consists of swapping elements of the pivot row if swapping is needed. Its completion section is again similar to the completion section of the pivot search phase and the only difference is that the process which will execute it will setup the row reduction phase and make it the next ready phase.

The row reduction phase is organized similarly. Its computational section consists of DOALL loop that will perform the reduction of rows. Again, the completion section preserves the general structure seen in the previous two phases, with the exception that the process which executes this section will check to see if the LU factorization has been completed. If it is, all phases are marked as completed and an exit flag is set so that processes will exit the subroutine, otherwise, the row marker (pivot counter) is incremented and the three phases are repeated until the LU factorization is complete. The outline of this parallel algorithm is as shown in Table II. The performance of the implemented LU factorization algorithm, on Encore's Multimax, Alliant's FX/8 and Sequent's Balance 8000, is shown in Figures (4,5,6), respectively. In these figures, the performance of the new algorithm is the same as Jordan's when multiprogramming is not an issue. When it is, however, its effect is apparent, but the performance of the algorithm is far superior to the performance of Jordan's algorithm. The light bars show the execution time of the new LU algorithm with no other program running on the machine other than this job. In this case, on Alliant's and Sequent's machines, running with up to 12 processes is almost 80% slower than running with the minimum execution time with 8 processes but is still less than 2/3 of the single process time. Similarly, on a 20 processor Encore running with 24 is less than 1/3 of the single process time. The dark bars represent the execution time in an artificially loaded environment. In this case the minimum time was obtained when running with 7 processes on the

Multimax and running with 3 processes on Alliant's and Sequent's machines. On all of these machines the execution time is less sensitive to the number of processes exceeding the number of available processors. Thus, there is a better balance between individual parallel program performance and overall system utilization.

4. Conclusions In this paper we presented a new programming methodology that can reduce inefficiencies that may result from running parallel programs on multiprogrammed shared memory multiprocessors. On these machines, the performance of parallel programs that are tightly synchronized may be severely affected and running with more than one process will result in a real execution time that is slower than running with a single process. Our methodology reduces this effect and depends on work completions as the basis for progress in parallel programs. This methodology will take into account the possibility of swapping processes by avoiding pre-scheduling of computations and avoiding the use of barriers that tightly synchronize instruction streams.

REFERENCES

(1) H. F. Jordan, "The Force," *in The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon and R. J. Douglass, Eds., Chap. 16, MIT Press (1987).

(2) G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing, August 1985.*

(3) W. Cowther, J. Goodhue, E. Starr, R. Thomas, W. Williken and T. Blackadar, "Performance Measurements on a 128-Node Butterfly Parallel Processor," *Proceedings of the 1985 International Conference on Parallel Processing, August 1985.*

(4) J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users Guide*, SIAM Pub., Phil., PA (1979).

(5) W. H. Jones, "Increasing Processor Utilization During Parallel Computation Rundown," *Proceedings of the 1986 International Conference on Parallel Processing, August 1986.*

(6) H. F. Jordan, "Structuring parallel algorithms in an MIMD, shared memory environment," *Parallel Computing, Vol. 3, No. 2, pp. 93-110, May 1986.*

(7) H. F. Jordan, M. S. Benten and N. S. Arenstrof, "Force User's Manual," *ECE Tech. Rept. CSDG 86-1-4, Computer System Design Group, Electrical and Computer Engineering Department, University of Colorado, Boulder, Oct 1986.*

(8) N. S. Arenstrof and H. F. Jordan, "Comapring Barrier Algorithms," *ECE Tech. Rept. CSDG 87-1-2, Computer System Design Group, Electrical and Computer Engineering Department, University of Colorado, Boulder, June 1987.*

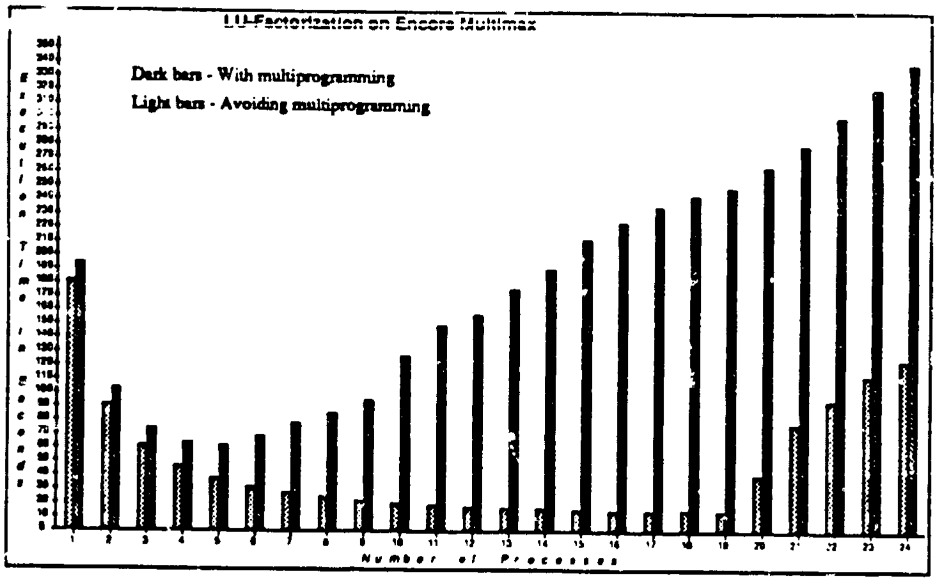
(9) C. D. Polychronopoulos, D. J. Kuck and D. A. Padua, "Execution of Parallel Loops on Parallel Processor Systems," *Proceedings of the 1986 International Conference on Parallel Processing, August 1986.*

TABLE-I Jordan's LU Factorization Of a Matrix

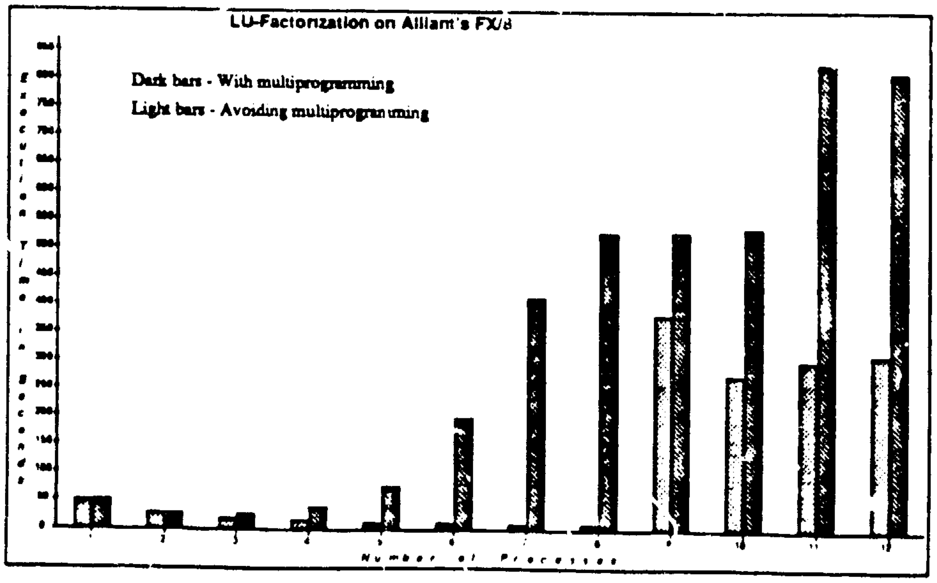
```
Barrier
 Initialize;
End barrier;

for k=1,......,n-1 do
  begin

   DOALL i=k,.....n-1
    find l such that ABS( $a_{l,k}$ ) > max ( $a_{i,k}$ );

   Critical
    If l > ipvt(k) then
      save l in ipvt(k);

Barrier
End barrier;

   If l <> k
   DOALL j=k,... n-1
    interchange $a_{k,j}$ and $a_{l,j}$;

   Barrier
    piv = -1.0 / $a_{k,k}$;
   End barrier;

   DOALL i=k+1,.....,n
     begin
      $m_i$ = -$a_{i,k}$ * piv
      for j=k+1,.....n do
        $a_{i,j}$ = $a_{i,j}$ + $m_i$ * $a_{k,j}$;

     end i;

  Barrier
     Reinitialize;
  End barrier;
 end k.
```

TABLE-II The New LU Factorization Of a Matrix

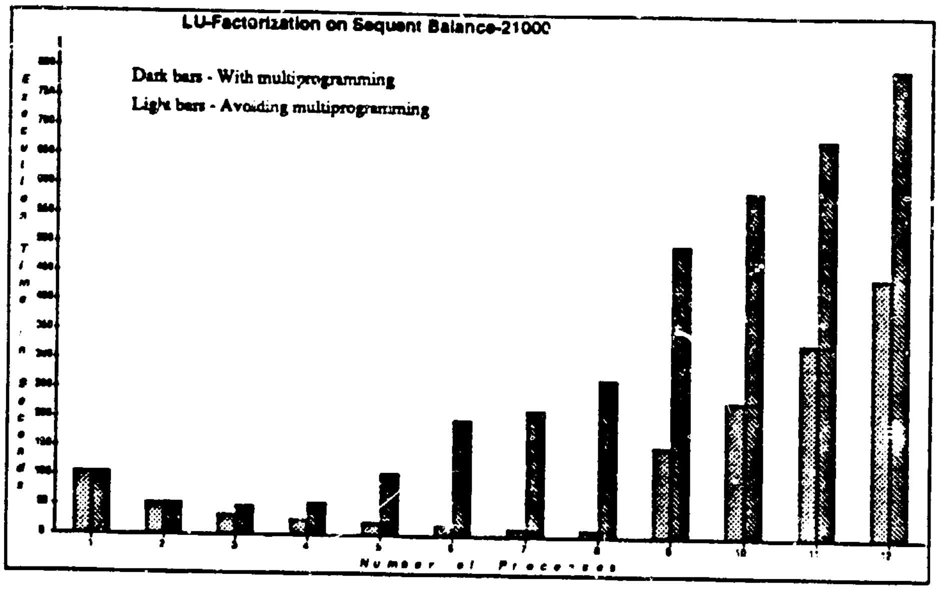
```
Init- (if not done)
 Initialize;

L1:  p1: Phase-1  (Ready , Blocked , Completed)

        DOALL i=k,.....n-1
         find l such that ABS( $a_{l,k}$ ) > max ( $a_{i,k}$ );

        Critical
         If l > ipvt(k) then
          save l in ipvt(k);

        Check-out (if not done and work completed)
          begin
           Set Phase-1 blocked
           Setup Phase-2 and mark it ready
          end;
       End Phase-1

     p2: Phase-2    (Ready , Blocked , Completed)
        DOALL j=k,.....n-1
         interchange $a_{k,j}$ and $a_{l,j}$;

        Check-out (if not done and work completed)
          begin
           Set Phase-2 blocked
           piv = -1.0 / $a_{k,k}$;
           Setup Phase-3 and mark it ready
          end;
       End Phase-2

     p3: Phase-3    (Ready , Blocked , Completed)
        DOALL i=k,.....n-1
          $m_i = -a_{i,k} * piv$
          for j=k+1,....n do
            $a_{i,j} = a_{i,j} + m_i * a_{k,j}$;

        Check-out (if not done and work completed)
          begin
           Set Phase-3 blocked
           if (Factorization Complete)
             begin
              Set Phase-1 completed
              Set Phase-2 completed
              Set Phase-3 completed
             end
           else
              Reinitialize
              Setup Phase-1 and mark it ready
          end;
      End Phase-3
 If (factorization not complete) goto L1
end .
```



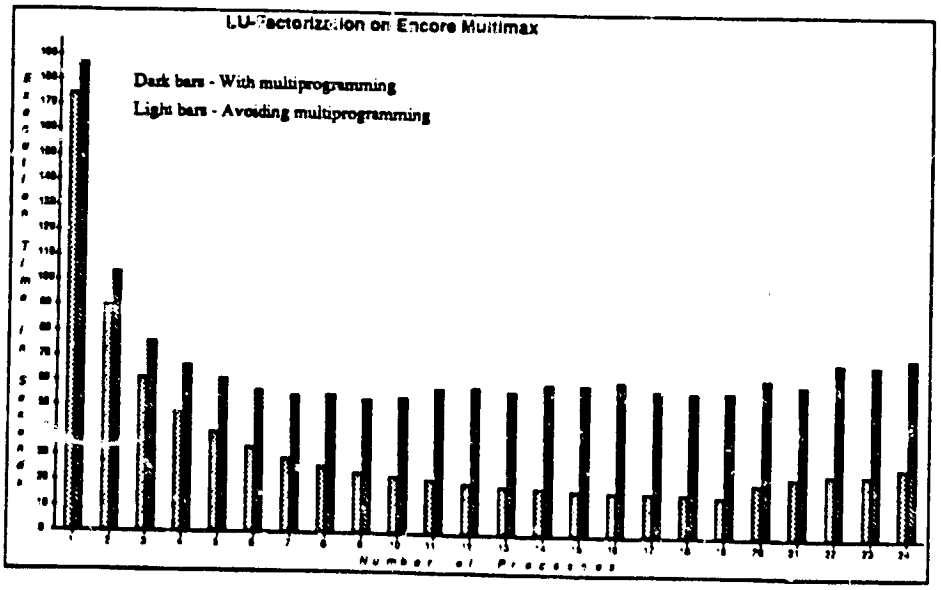
Figure(1) LU-Factorization on Encore's Multimax using Barriers



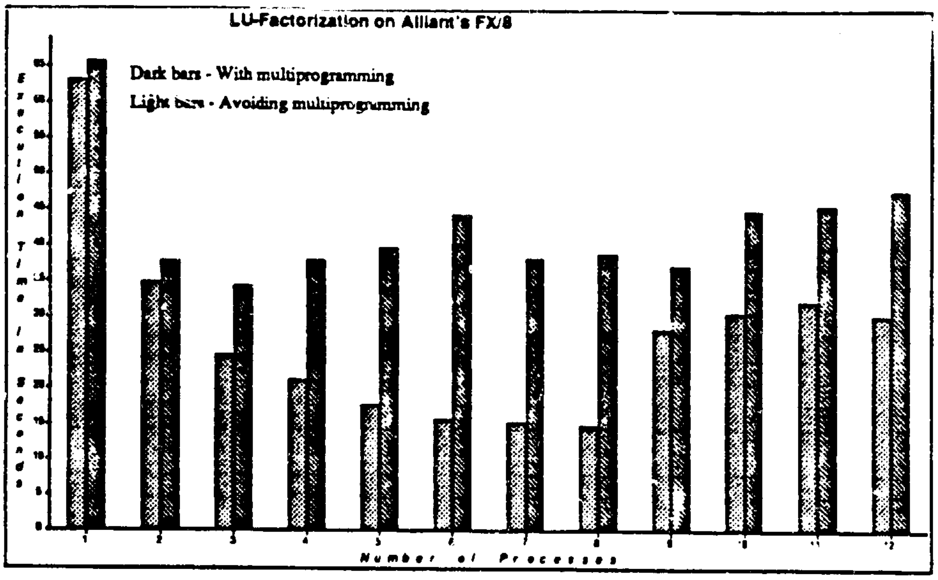
Figure(2) LU-Factorization on Alliant's FX/8 using Barriers



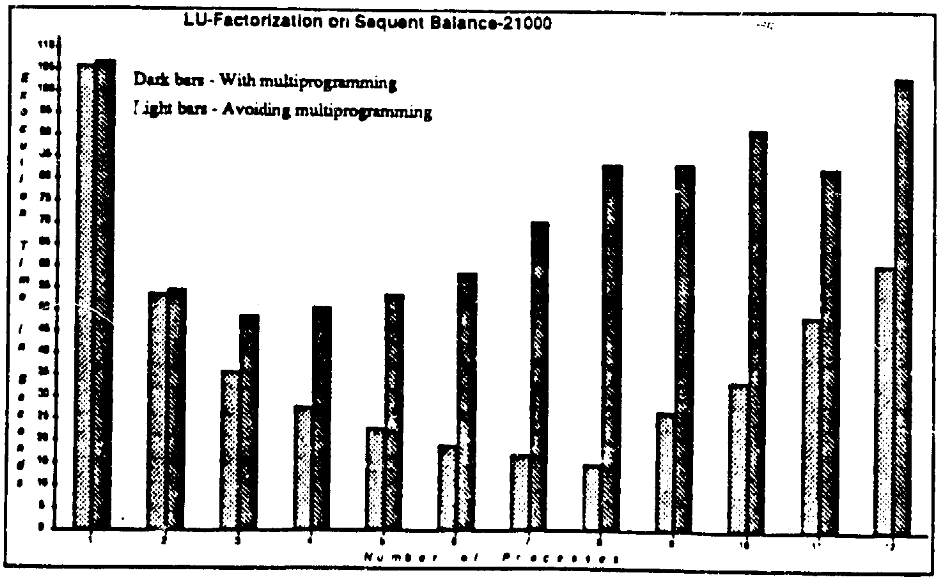
Figure(3) LU-Factorization on Sequent's Balance-21000 using Barriers



Figure(4) LU-Factorization on Encore's Multimax using the new method



Figure(5) LU-Factorization on Alliant's FX/8 using the new method



Figure(6) LU-Factorization on Sequent's Balance-21000 using the new method